CMPE 365 Project – Recurrence Problem

This project was completed using C++11.

For future reference:

unordered\_map in C++11 is essentially a hash table.

M is the number of planes.

N is the number of unique people.

P is the average plane size.

# Algorithm Options

Two algorithms were examined as possible solutions for this problem.

## Searching

For each plane, this algorithm iterates through each pair in the plane using nested for loops. A table is used to count the number of times each pair occurs, using each person’s ID – 1 as the location to increment. When the value at this table location is at the threshold, the pair is inserted into an unordered\_map with the person IDs concatenated as a string as the key and the count as the value. This is a unique key as the person IDs are all unique. When that pair key is already in the map, the value at that location is incremented so that the map contains the actual number of times the pair occurred, for completeness.

After each plane is computed, the unordered\_map is simply iterated through to output each pair that re-occurred the given number of times (or greater).

## Marking

This algorithm uses the memoization concept from dynamic programming with respect to who is present within a given plane. Rather than iterating through each plane in nested for loops, a plane is iterated through once, marking which people are present in a table. This table is simply a matrix, initialized at 0s, of each possible person ID – 1 (ID’s are unique, -1 as arrays in C++ are indexed at 0). Marking is simply incrementing the value at markingTablle[i][personID-1] and at markingTable[personID-1][i]. When i is equal to personID-1, no marking is done as this would be the same person in the pair. Should an entry in the table have 2 after a new marking, that means there is a pair present in the plane. An unordered\_map is used to store that pair as a key (string concatenated, unique as the person IDs are unique), with the value as 1 (the number of times they have been on a plane together. If the map already holds that key, instead of insertion, the value at that key is incremented.

This is done for each plane with a newly zeroed marking matrix. Therefore, after all planes have been computed, the unordered\_map holds all recurring pairs and the number of times they re-occur. The map is then iterated through to find the pairs, based on the key value, that re-occur the given number of times (or greater).

To avoid extra compute time of searching for markings of 2 after marking is over, each time a marking is made, that newly updated location is checked for a marking of 2. If it has a marking of 2 it is immediately inserted into the unordered\_map.

# Algorithm Complexity

Clearly, any solution to this problem must examine each person on each plane at least once. That means the absolute optimal solution must have a complexity of O(MP) or worse. This is useful to compare with the created algorithm complexities.

Since there are N people, P must be less than or equal to N.

## Searching

## Marking

## Comparison

The searching algorithm complexity is O(MP2) whereas the marking algorithm complexity is O(MPN). Since P <= N, this suggests the searching algorithm has a slightly better complexity.

Technically, P is a function of N. P will be N/K, where K is a constant. For example, a decent guess would be P is N/2. Therefore, O(P) is really O(N/K), which falls to O(N). Thus one can argue the Big-O of the algorithms are actually both O(MN2). This is true for the worst possible case of P = N, although we know in practice that is not true and the searching algorithm has slightly better Big-Theta and Big-Omega complexities.

# Results

## Correctness

## Speed

# Appendix A

## For Loop Speed

According to research, in C++11 in the manual for loop

for (int i=0; i<myVector.size(); ++I)

Calls size() every loop iteration. In the range based for loop

for (const auto &p : myVector)

Creates a copy of the iterator end() of myVector. This cached copy is used for comparison and therefore is more efficient than the manual for loop. Also, the use of **const auto** & improves performance. In C++11, a constant reference is more quickly created and changed than initializing an int variable and using the assignment operator to change its value. The use of **auto** helps avoid implicit type conversion, improving performance.

I would have expected these differences to be nullified by compiler optimizations, or at least have relatively negligible effects on the execution time. However, I tested the execution time of the following pieces of code:

for (const auto &plane : planes) {

for (const auto &person : plane) {

for (int i=0; i<NUM\_PPL; ++i);

}

}

for (const auto &plane : planes) {

for (int i=0; i<plane.size()-1; ++i) {

for (int j=i+1; j<plane.size(); ++j);

}

}

The first piece of code is used in the marking algorithm, the second in the searching algorithm. The respective approximate execution times for just these loops are 7 milliseconds and 25000 milliseconds. This difference seems extreme to me, but these tests have been quite consistent on my machine and thus I will accept them. Clearly, this difference means the marking algorithm should be used.

## Complexity of to\_string()

## Complexity of Matrix Initialization